COST-EFFECTIVE DATA-PARALLEL LOAD BALANCING

James P. Ahrens
Dept. of Computer Science & Engineering
University of Washington
Seattle, Washington 98195
ahrens@cs.washington.edu

Charles D. Hansen
Advanced Computing Laboratory
Los Alamos National Laboratory
Los Alamos, New Mexico 87545
hansen@acl.lanl.gov

Abstract – Load balancing algorithms improve a program's performance on unbalanced datasets, but can degrade performance on balanced datasets, because unnecessary load redistributions occur. This paper presents a cost-effective data-parallel load balancing algorithm which performs load redistributions only when the possible savings outweigh the redistribution costs. Experiments with a data-parallel polygon renderer show a performance improvement of up to a factor of 33 on unbalanced datasets and a maximum performance loss of only 27 percent on balanced datasets when using this algorithm.

1. INTRODUCTION

Load balancing algorithms provide the basis for efficient parallel solutions to many important computational problems including the n-body problem, polygon and volume rendering, and optimization problems. The completion time of these parallel solutions depends on the completion time of the processor with the maximum computational workload. Load balancing algorithms attempt to distribute the computational workload evenly among all processors. This reduces the maximum workload on any processor and thus reduces the completion time of the parallel solutions.

A significant problem when using a load balancing algorithm is the possibility that along with improving performance on some datasets it will degrade performance on others. Wikstrom et al. [1] use a computation model and experimental results to present evidence that using a load balancing algorithm does not always improve a program's performance. The authors show the execution time of a load-balanced version of a program can substantially exceed the execution time of the original version of the same program. This is because the costs of load redistributions can exceed the savings achieved by the redistributions.

Other researchers have studied the problem of deciding when to balance with different workloads and problem types. Nicol and Townsend [2] describe a partitioning strategy which uses performance measurements to decide how to partition an irregular grid among processors. Nicol and Reynolds [3] describe a data-parallel load balancing algorithm which is targeted for applications with uncertain behavior. The algorithm uses a probabilistic model of the cost of delay and the benefits of balancing to decide when to run a single balancing operation.

In this paper, a load balancing algorithm is described which uses a prediction of the costs and calculation of the possible savings to decide when redistribution is cost-effective. A major advantage of utilizing a cost-effective load balancing algorithm is that the execution time of a load-balanced version of a program is never significantly worse than the execution time of the original version of the same program. This result depends only on never underestimating the costs of load balancing.

In Section 2, the type of workloads and programs the load balancing algorithm works with is presented along with a high-level description of the load balancing algorithm. Section 2 also describes when and how load balancing occurs. Section 3 presents a performance study of a data-parallel rendering program to which the load balancing algorithm has been applied. Section 4 concludes.

2. THE LOAD BALANCING ALGORITHM

The load balancing algorithm can be applied to data-parallel programs which compute the solutions of a collection of independent tasks. The tasks are independent in time; they do not have to execute in any specific order and there are no data dependencies between the tasks. A set of tasks can share the same read-only problem data. An example of how a set of tasks share the same problem data occurs in the polygon rendering application. Multiple tasks are used to process a row of pixels. Each task computes the solution for one pixel in the row. The pixel's location is computed by adding an offset to the row's initial pixel location. The initial pixel location is stored as part of the problem data.

For the following discussion, we assume a virtual processing facility provides the abstraction of having one virtual processor assigned to each data element of a parallel array. In a prototypical program, each processor is assigned problem data and its associated tasks. A processor's workload is the number of tasks associated with its assigned problem data. Values in the range $1..number\_of\_tasks$ are used as indices to refer to these tasks. A task's index is used to calculate the specific problem data on which the task computes. In the pixel processing example described above, the task index is used as an offset; task i processes the ith pixel location. The prob-

```
<initial instructions>
forloop index = 1, MAX(workload) {
    WHERE (index ≤ workload) {
        Solution Phase(index)}}
<further instructions>
```

Figure 1: A prototypical program to be augmented with the load balancing algorithm

lem data and workload are stored in parallel arrays named *problem_data* and *workload*.

In order to process the tasks, the program increments a global task index counter, *index*, which starts at 1 and ends at the maximum workload of all the processors. During each iteration of the global index, each processor checks if they have a task with that index, and if they do, they compute a solution for the task. The instructions used to compute the solution of a multiple tasks in parallel are called the **solution phase**. A pseudo-code description of a prototypical program is shown in Figure 1. Note that the WHERE statement activates processors for which the test is true and idles processors elsewhere.

As the computation proceeds, more and more processors complete the processing of their tasks and remain idle for the rest of the loop iterations. These processors are then termed *idle* processors. Processors which have tasks to complete are termed *active* processors. Processors are idled because all processors must process tasks with the same task index at the same time.

To improve the program's performance, the program is modified so that tasks with different indices can be processed at the same time. The load balancing algorithm then distributes tasks from heavily loaded active processors to idle processors and tries to balance the workload among all processors.

How a program is augmented with the load balancing algorithm is now described. The load balancing algorithm consists of three distinct phases: the **information gathering phase**, the **decision phase** which decides when load balancing should occur and the **redistribution phase** which distributes tasks from active processors to idle processors. The basic iteration structure of the program is preserved. At the beginning of each iteration, the information gathering phase is executed. Then the decision phase is run, utilizing the gathered information to decide if balancing should occur during this iteration. If the decision is to balance, the redistribution phase is run, moving problem data from active to idle processors and assigning these idle processors new task indices to process. When tasks are distributed, the task indices originally assigned to an active processor can be assigned to multiple idle processors. Thus, different processors can work on different task indices during the same iteration. In the pixel processing example, this could mean, for example, that the first 3 pixels of one processor's row are processed along with the first 5

```
<initial instructions>
loop {
    Information Gather Phase
    IF (Decision Phase returns TRUE) THEN {
        Redistribution Phase }
    WHERE (workload > 0) {
        Solution Phase(parallel_index)
        workload = workload - 1 }}
until (All elements of workload = 0)
<further instructions>
```

Figure 2: A prototypical load-balanced program

pixels of another processor's row. A parallel array, named *parallel_index*, is used to keep track of the current task index computed by each processor. A pseudo-code description of a prototypical load-balanced program is shown in Figure 2.

Note that the execution time of both the original and load-balanced prototypical programs presented in Figures 1 and 2 is proportional to the maximum number of tasks assigned to any processor. The major difference between the programs, is the load-balanced program can reduce the maximum number of tasks on any processor by redistributing the workload.

2.1 When to load balance

The information gathering phase creates a trial workload, named *new_workload*, which is used by the decision phase to decide if redistributing load will be cost-effective. This new workload contains a more balanced distribution of tasks and therefore has a smaller maximum number of tasks on any processor than the original workload. From this new workload a measure of the possible savings is calculated as the maximum number of tasks on any processor in the original workload minus the maximum number of tasks on any processor in the new workload. This is shown in the equation below:

$$\textit{savings} \text{ in iterations} = MAX(workload) - MAX(new\_workload)$$

The savings are measured in terms of the number of future iterations of the loop which will execute the solution phase. The maximum number of tasks in a workload dictates the number of iterations that must be executed to process these tasks. If the new distribution is used then these "saved" iterations will not have to be executed.

The costs of the load balancing algorithm are also measured. Since the savings are measured in terms of the number of iterations, the costs are converted to this unit as well. The costs of the load balancing algorithm are incurred during the execution of the information gathering phase and redistribution phase. In order to quantify the costs of these phases, during each iteration their execution times are measured. The execution time of the solution phase during each iteration is also measured. $time_{info}$, $time_{redis}$ and $time_{soln}$ are the execution times of one execution of the

information gathering, redistribution and solution phases. An estimate of the load balancing cost in terms of number of iterations can then be calculated by multiplying the sum of the execution time of the information gathering and redistribution phases by the inverse of the execution time of the solution phase, as shown in the equation below:

$$costs \text{ in iterations } = (time_{info} + time_{redis}) \times \frac{1 \text{ iteration}}{time_{soln}}$$

In order to provide a guarantee that the load balancing algorithm will always make cost-effective load balancing decisions, this cost measure must not be underestimated. Initial runs of the load-balanced program on various datasets are used to compute an overestimated cost measurement. The longest information gathering time of any iteration and redistribution time of any iteration are then divided by the shortest solution time of any iteration for each dataset. The largest of the resulting cost measures provides an estimate of an upper bound on the load balancing cost in terms of iterations. To this initial estimate a constant is added to assure the cost measure will always be an overestimate. This overestimate is then used in all future runs of the load balanced program.

Utilizing the overestimated costs and the calculated savings a cost-effective load balancing decision is then made by the decision phase. If the savings are greater than the costs then the redistribution phase is executed. Since each load balancing decision results in a cost-effective iteration, the sum of these decisions results in a cost-effective program execution.

2.2 How to load balance

An efficient redistribution algorithm is essential for good performance. Biagioni and Prins [4] and Nicol [5] describe efficient data-parallel redistribution algorithms which use scan communication routines to organize data movement. Our algorithm also uses scans to organize data movement. In our redistribution algorithm, workload is distributed by copying problem data from heavily loaded active processors to idle processors. The workload, in the form of task indices, is then divided up and assigned to the active and idle processors with copies of the problem data. Each active processor's workload is assigned some number of idle processors. This assignment is computed by assigning the idle processors in proportion to the workload on each active processor. Thus, heavy workloads are assigned more idle processors than light workloads and load is balanced evenly. A more detailed description of the redistribution algorithm is presented in [6].

3. A PERFORMANCE STUDY

The load balancing algorithm has been added to a data-parallel polygon renderer [7]. A series of experiments were executed using the original and a load balanced version of the renderer. The steps taken by the renderer include: *scan conversion*, which maps the polygons onto the rows (*scan lines*) of the resulting image and *z-buffering*, which maps the scan lines into pixels of the image. In the original version of the renderer, the scan conversion and z-buffering steps were implemented using data-parallel loops of the form shown in Figure 1. In the load balanced version, the steps were implemented using load balanced data-parallel loops of the form shown in Figure 2. For the scan conversion loop, the problem data is polygons and each task consists of creating and processing a scan line from a polygon. The number of scan lines in a polygon is dependent upon its image-space height. In the z-buffering loop, the problem data is scan lines and each task consists of creating and processing pixels from a scan line. The number of pixels in a scan line is dependent upon its length. For all the experiments, the renderer generates output images which are 512×512 pixels in size. The experiments were executed on the Advanced Computing Laboratory's 1024 processor CM-5 at Los Alamos National Laboratory.

Table 1: The Maximum and Average Workloads for the Scan Conversion Loop for the Balanced and Unbalanced Datasets

	Balanced		Unbalanced	
View	Max	Avg	Max	Avg
(0,0)	8	5	231	6
(45,45)	10	5	169	6
M(0,0)	13	5	512	8
M(45,45)	21	1	468	4

In the first experiment, performance data for the original and load balanced renderers is presented. The polygon datasets used in this experiment are a balanced and unbalanced version of the same scientific output, a hydrodynamics simulation of an oil well perforator. The relative balance of a polygon dataset is dependent upon the type of algorithm used to generate the dataset, the viewing and magnification transformations applied to the dataset, and the number of its polygons which have been clipped from view. Table 1 presents the maximum and average workloads of these datasets for the scan conversion loop. The first column of the table lists the viewing and magnification transformations that were applied to the datasets. The "M" before the viewing angle means the dataset has been magnified. Notice in the balanced datasets the difference between the maximum and average workload is small, whereas in the unbalanced datasets the difference is large.

On the unbalanced datasets, the decision phase and the redistribution phase work together to effectively to improve the renderer's performance. Table 2 shows the results of rendering the unbalanced polygons with (**LB**) and without (**OR**) the assistance of the load balancing algorithm on 32, 64, 128, 256 and 512 processors of the CM-5. Notice the poor performance of the original renderer on the unbalanced datasets and the improvement obtained when

Table 2: Rendering of Unbalanced Datasets in Seconds

	32		64		128		256		512	
View	OR	LB	OR	LB	OR	LB	OR	LB	OR	LB
(0,0)	48.91	6.21	28.98	3.57	18.18	2.12	12.76	1.35	9.99	0.92
(45,45)	38.03	5.33	22.37	3.00	14.27	1.78	10.45	1.15	8.24	0.81
M (0,0)	109.64	10.62	65.50	5.90	41.74	3.40	30.12	2.16	23.92	1.46
M (45,45)	99.13	4.42	57.91	2.42	37.19	1.46	27.33	0.94	22.40	0.67

Table 3: Rendering of Balanced Datasets in Seconds

	32		64		128		256		512	
View	OR	LB	OR	LB	OR	LB	OR	LB	OR	LB
(0,0)	4.71	5.16	2.67	2.95	1.57	1.72	0.88	1.00	0.54	0.64
(45,45)	5.49	6.12	2.98	3.33	1.69	1.89	0.99	1.13	0.62	0.73
M (0,0)	10.61	12.47	5.80	6.77	3.18	3.73	1.82	2.21	1.11	1.40
M (45,45)	12.27	13.43	6.69	7.34	3.74	4.12	2.24	2.50	1.51	1.65

using the load balancing algorithm. The performance of the load-balanced renderer provides a factor of 8 to 33 improvement over the performance of the original renderer on the unbalanced datasets.

Table 3 shows the results of rendering the balanced polygons with and without the assistance of the load balancing algorithm on 32, 64, 128, 256 and 512 processors. Notice that when the load balanced renderer is applied to the balanced datasets its performance is approximately the same as the original renderer. It is difficult for a load balancing algorithm to provide good performance on a balanced dataset since any redistribution steps will simply waste time. The worst case empirical performance loss is only 27 percent on balanced datasets when using the load balancing algorithm.

The original renderer's performance on the balanced datasets provides an estimate of the target performance we would like to achieve with the addition of a load balancing algorithm. The performance of the load-balanced renderer on the unbalanced datasets is within 70 percent of the performance of the original renderer on the balanced datasets.

In a second experiment, three other polygon datasets were tested. Two datasets were generated from different outputs of a fluid-dynamics simulation and the other from the output of a particle interaction simulation. Two of the the datasets are balanced and one is unbalanced. In summary, performance improvements ranged from a factor of 4 to 33 and the worst case empirical performance loss is only 25 percent.

4. CONCLUSIONS

A significant problem when using a load balancing algorithm is the possibility that along with improving performance on some datasets it will degrade performance on others. In this paper, a data-parallel load balancing algorithm was described which will not substantially degrade a program's performance on any dataset. This property results from utilizing an empirical measurement of the cost of load balancing along with a calculation of the possible savings to restrict load balancing to only when it is cost-effective.

Acknowledgments This research was performed at the Advanced Computing Laboratory of Los Alamos National Laboratory, Los Alamos, NM 87545.

REFERENCES

[1] M. C. Wikstrom, G. M. Prabhu, and J. L. Gustafson. Myths of load balancing. In *Parallel Computing '91*, pages 531–549, 1991.

[2] D. M. Nicol and J. C. Townsend. Accurate modeling of parallel scientific computation. In *Proceedings of the 1989 SIGMETRICS Conference*, pages 165–170, May 1989.

[3] D. M. Nicol and P. F. Reynolds Jr. Optimal dynamic remapping of data parallel computations. *IEEE Transactions on Computers*, 39(2):206–219, February 1990.

[4] E. S. Biagioni and J. F. Prins. Scan directed load balancing for highly parallel mesh-connected parallel computers. In *Unstructured Scientific Computation on Scalable Multiprocessors*, pages 371–95, October 1990.

[5] D. M. Nicol. Communication efficient global load balancing. In *Proceedings of the Scalable High Performance Computing Conference*, pages 292–299, April 1992.

[6] J. P. Ahrens and C. D. Hansen. Cost-effective data-parallel load balancing. Technical Report TR-95-04-02, University of Washington, 1995.

[7] F. A. Ortega, C. D. Hansen, and J. P. Ahrens. Fast data parallel polygon rendering. In *Proceedings of Supercomputing '93*, pages 709–718, November 1993.